LEARN C++
Skill up with our free tutorials

2.9 — Naming collisions and an introduction to namespaces

ALEX[1] MARCH 18, 2024

Let's say you are driving to a friend's house for the first time, and the address given to you is 245 Front Street in Mill City. Upon reaching Mill City, you take out your map, only to discover that Mill City actually has two different Front Streets across town from each other! Which one would you go to? Unless there were some additional clue to help you decide (e.g. you remember your friend's house is near the river) you'd have to call your friend and ask for more information. Because this would be confusing and inefficient (particularly for your mail carrier), in most countries, all street names and house addresses within a city are required to be unique.

Similarly, C++ requires that all identifiers be non-ambiguous. If two identical identifiers are introduced into the same program in a way that the compiler or linker can't tell them apart, the compiler or linker will produce an error. This error is generally referred to as a **naming collision** (or **naming conflict**).

If the colliding identifiers are introduced into the same file, the result will be a compiler error. If the colliding identifiers are introduced into separate files belonging to the same program, the result will be a linker error.

An example of a naming collision

a.cpp:

```
#include <iostream>

void myFcn(int x)
{
    std::cout << x;
}
```

main.cpp:

```
#include <iostream>

void myFcn(int x)
{
    std::cout << 2 * x;
}

int main()
{
    return 0;
}
```

When the compiler compiles this program, it will compile *a.cpp* and *main.cpp* independently, and each file will compile with no problems.

However, when the linker executes, it will link all the definitions in *a.cpp* and *main.cpp* together, and discover conflicting definitions for function `myFcn()`. The linker will then abort with an error. Note that this error occurs even though `myFcn()` is never called!

Most naming collisions occur in two cases:

1. Two (or more) identically named functions (or global variables) are introduced into separate files belonging to the same program. This will result in a linker error, as shown above.
2. Two (or more) identically named functions (or global variables) are introduced into the same file. This will result in a compiler error.

As programs get larger and use more identifiers, the odds of a naming collision being introduced increases significantly. The good news is that C++ provides plenty of mechanisms for avoiding naming collisions. Local scope, which keeps local variables defined inside functions from conflicting with each other, is one such mechanism. But local scope doesn't work for function names. So how do we keep function names from conflicting with each other?

Scope regions

Back to our address analogy for a moment, having two Front Streets was only problematic because those streets existed within the same city. On the other hand, if you had to deliver mail to two addresses, one at 245 Front Street in Mill City, and another address at 417 Front Street in Jonesville, there would be no confusion about where to go. Put another way, cities provide groupings that allow us to disambiguate addresses that might otherwise conflict with each other.

A **scope region** is an area of source code where all declared identifiers are considered distinct from names declared in other scopes (much like the cities in our analogy). Two identifiers with the same name can be declared in separate scope regions without causing a naming conflict. However, within a given scope region, all identifiers must be unique, otherwise a naming collision will result.

The body of a function is one example of a scope region. Two identically-named identifiers can be defined in separate functions without issue -- because each function provides a separate scope region, there is no collision. However, if you try to define two identically-named identifiers within the same function, a naming collision will result, and the compiler will complain.

Namespaces

A **namespace** provides another type of scope region (called **namespace scope**) that allows you to declare names inside of it for the purpose of disambiguation. Any names declared inside the namespace won't be mistaken for identical names in other scopes.

Key insight

A name declared in a scope region (such as a namespace) won't be mistaken for an identical name declared in another scope.

Unlike functions (which are designed to contain executable statements), only declarations and definitions can appear in the scope of a namespace. For example, two identically named functions can be defined inside separate namespaces, and no naming collision will occur.

Key insight

Only declarations and definitions can appear in the scope of a namespace (not executable statements). However, a function can be defined inside a namespace, and that function can contain executable statements.

Namespaces are often used to group related identifiers in a large project to help ensure they don't inadvertently collide with other identifiers. For example, if you put all your math functions in a namespace named `math`, then your math functions won't collide with identically named functions outside the `math` namespace.

We'll talk about how to create your own namespaces in a future lesson.

The global namespace

In C++, any name that is not defined inside a class, function, or a namespace is considered to be part of an implicitly-defined namespace called the **global namespace** (sometimes also called **the global scope**).

In the example at the top of the lesson, functions `main()` and both versions of `myFcn()` are defined inside the global namespace. The naming collision encountered in the example happens because both versions of `myFcn()` end up inside the global namespace, which violates the rule that all names in the scope region must be unique.

We discuss the global namespace in more detail in lesson 7.4 -- Introduction to global variables (https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/)[2].

For now, there are two things you should know:

- Identifiers declared inside the global scope are in scope from the point of declaration to the end of the file.
- Although variables can be defined in the global namespace, this should generally be avoided (we discuss why in lesson 7.8 -- Why (non-const) global variables are evil[3]).

For example:

```
#include <iostream> // imports the declaration of std::cout into the global scope

// All of the following statements are part of the global namespace

void foo();    // okay: function forward declaration
int x;         // compiles but strongly discouraged: non-const global variable definition (without initializer)
int y { 5 };   // compiles but strongly discouraged: non-const global variable definition (with initializer)
x = 5;         // compile error: executable statements are not allowed in namespaces

int main()     // okay: function definition
{
    return 0;
}

void goo();    // okay: A function forward declaration
```

The std namespace

When C++ was originally designed, all of the identifiers in the C++ standard library (including std::cin and std::cout) were available to be used without the `std::` prefix (they were part of the global namespace). However, this meant that any identifier in the standard library could potentially conflict with any name you picked for your own identifiers (also defined in the global namespace). Code that was working might suddenly have a naming conflict when you #included a new file from the standard library. Or worse, programs that would compile under one version of C++ might not compile under a future version of C++, as new identifiers introduced into the standard library could have a naming conflict with already written code. So C++ moved all of the functionality in the standard library into a namespace named `std` (short for "standard").

It turns out that `std::cout`'s name isn't really `std::cout`. It's actually just `cout`, and `std` is the name of the namespace that identifier `cout` is part of. Because `cout` is defined in the `std` namespace, the name `cout` won't conflict with any objects or functions named `cout` that we create outside of the `std` namespace (such as in the global namespace).

When accessing an identifier that is defined in a namespace (e.g. `std::cout`), you need to tell the compiler that we're looking for an identifier defined inside the namespace (`std`).

Key insight

When you use an identifier that is defined inside a namespace (such as the `std` namespace), you have to tell the compiler that the identifier lives inside the namespace.

There are a few different ways to do this.

Explicit namespace qualifier std::

The most straightforward way to tell the compiler that we want to use `cout` from the `std` namespace is by explicitly using the `std::` prefix. For example:

```
#include <iostream>

int main()
{
    std::cout << "Hello world!"; // when we say cout, we mean the cout defined in the std namespace
    return 0;
}
```

The :: symbol is an operator called the **scope resolution operator**. The identifier to the left of the `::` symbol identifies the namespace that the name to the right of the `::` symbol is contained within. If no identifier to the left of the `::` symbol is provided, the global namespace is assumed.

So when we say `std::cout` we're saying "the `cout` that is declared in namespace `std` ".

This is the safest way to use `cout`, because there's no ambiguity about which `cout` we're referencing (the one in the `std` namespace).

Best practice

Use explicit namespace prefixes to access identifiers defined in a namespace.

When an identifier includes a namespace prefix, the identifier is called a **qualified name**.

Using namespace std (and why to avoid it)

Another way to access identifiers inside a namespace is to use a using-directive statement. Here's our original "Hello world" program with a using-directive:

```
#include <iostream>

using namespace std; // this is a using-directive that allows us to access names in the std namespace with no namespace prefix

int main()
{
    cout << "Hello world!";
    return 0;
}
```

A **using directive** allows us to access the names in a namespace without using a namespace prefix. So in the above example, when the compiler goes to determine what identifier `cout` is, it will match with `std::cout`, which, because of the using-directive, is accessible as just `cout`.

Many texts, tutorials, and even some IDEs recommend or use a using-directive at the top of the program. However, used in this way, this is a bad practice, and highly discouraged.

Consider the following program:

```cpp
#include <iostream> // imports the declaration of std::cout into the global scope

using namespace std; // makes std::cout accessible as "cout"

int cout() // defines our own "cout" function in the global namespace
{
    return 5;
}

int main()
{
    cout << "Hello, world!"; // Compile error!  Which cout do we want here?  The one in the std namespace or the one we defined above?

    return 0;
}
```

The above program doesn't compile, because the compiler now can't tell whether we want the `cout` function that we defined, or `std::cout`.

When using a using-directive in this manner, *any* identifier we define may conflict with *any* identically named identifier in the `std` namespace. Even worse, while an identifier name may not conflict today, it may conflict with new identifiers added to the std namespace in future language revisions. This was the whole point of moving all of the identifiers in the standard library into the `std` namespace in the first place!

Warning

Avoid using-directives (such as `using namespace std;`) at the top of your program or in header files. They violate the reason why namespaces were added in the first place.

Related content

We talk more about using-declarations and using-directives (and how to use them responsibly) in lesson 7.12 -- Using declarations and using directives (https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/)[4].

Curly braces and indented code

In C++, curly braces are often used to delineate a scope region that is nested within another scope region (braces are also used for some non-scope-related purposes, such as list initialization). For example, a function defined inside the global scope region uses curly braces to separate the scope region of the function from the global scope.

In certain cases, identifiers defined outside the curly braces may still be part of the scope defined by the curly braces rather than the surrounding scope -- function parameters are a good example of this.

For example:

```
#include <iostream> // imports the declaration of std::cout into the global
scope

void foo(int x) // foo is defined in the global scope, x is defined within
scope of foo()
{ // braces used to delineate nested scope region for function foo()
    std::cout << x << '\n';
} // x goes out of scope here

int main()
{ // braces used to delineate nested scope region for function main()
    foo(5);

    int x { 6 }; // x is defined within the scope of main()
    std::cout << x << '\n';

    return 0;
} // x goes out of scope here
// foo and main (and std::cout) go out of scope here (the end of the file)
```

The code that exists inside a nested scope region is conventionally indented one level, both for readability and to help indicate that it exists inside a separate scope region.

The `#include` and function definitions for `foo()` and `main()` exist in the global scope region, so they are not indented. The statements inside each function exist inside the nested scope region of the function, so they are indented one level.

Next lesson
2.10 Introduction to the preprocessor

5

Back to table of contents

6

Previous lesson
2.8 Programs with multiple code files

7

8

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[10] are connected to your provided email address.

270 COMMENTS Newest

MBA
April 21, 2024 6:08 am

```
/* When accessing an identifier that is defined in a namespace (e.g.
std::cout), you need to tell the compiler that we’re looking for an
identifier defined inside the namespace (std).

Key insight

When you use an identifier that is defined inside a namespace (such as the
std namespace), you have to tell the compiler that the identifier lives
inside the namespace. */
```

Aren't these two lines exactly the same thing? I've seen a version of this repetition before this line too.

Last edited 3 days ago by MBA

0 Reply

Nutella
March 17, 2024 2:45 am

Hello, I have a question:

```
int y { 5 };   // compiles but strongly discouraged: non-const variable
definition with initializer
```

Why is it strongly discouraged?

0 Reply

Alex Author

Reply to Nutella 11 March 18, 2024 10:33 am

Non-const global variables are strongly discouraged.

1 Reply

IceFloe

February 20, 2024 8:16 pm

I understand that a namespace is a kind of container in which the identifiers and names we need are stored, and when using a prefix, how do we point to this container? But what if I want to create my own namespace list, it turns out that I have to use `using namespace userName;` ?

0 Reply

Alex Author

Reply to IceFloe 12 February 22, 2024 2:09 pm

The naming prefix identifies the namespace (or containing scope) the identifer is in.

We discuss creating your own namespaces in lesson https://www.learncpp.com/cpp-tutorial/user-defined-namespaces-and-the-scope-resolution-operator/

0 Reply

Swaminathan R

January 22, 2024 6:29 am

```
#include <iostream>
namespace A
{
    int main()
    { return 0;}
}

int main()
{
  std::cout<<A::main()<<'\n';
  std::cout<<"I work! But how? there are two mains?";
}
```

I expected this to error saying I can't have two functions with name "main()", but it compiles and runs just fine.

Another thing! out of curiosity, I tried this too. And it didn't throw any error (well, it doesn't error. But it keeps running for a while and shows nothing!) Can you let me know what happens here?

```
#include <iostream>
int main()
{
std::cout<<main();
}
```

0 Reply

Swaminathan
Reply to Swaminathan R 13 January 23, 2024 12:56 pm

Thank you Alex(s) ❤

0 Reply

Alex Author
Reply to Swaminathan R 13 January 23, 2024 10:17 am

The other Alex answered your first question.

Re: the latter, in C++ `main()` is not allowed to be called explicitly. However, if your compiler allows it, then this is a recursive function call (a function that calls itself) with no termination condition, so it will continue to call itself until the program runs out of stack space and is terminated.

3 Reply

Alex
Reply to Swaminathan R 13 January 23, 2024 6:42 am

You do not have two `main()` function. You have one `main()` and one `A::main()`

Second thing is just an UB. Don't do that.

2 Reply

tatadott
November 13, 2023 7:30 am

Would I be able to modify a integer variable declared in **some** namespace from **global** namespace?
Like this:
`some::variable = 1;`

Btw, thanks for the amazing lesson.

0 Reply

Alex Author

Reply to tatadott 14 November 14, 2023 9:09 pm

Yes.

1 Reply

tatadott

Reply to Alex 15 November 17, 2023 7:34 am

I misunderstood your reply.

I tried to compile this program by inserting `some::x = 1;` directly inside the global namespace which doesn't compile but it does compile if i modify it inside a function(`main` in this case) defined in global namespace. This led me to modify it in `some` namespace, but the remodified program also doesn't compile.

I think it would be good if you mention that statements except declaration and definition cannot appear at namespace scope in the `What is a namespace?` section instead of mentioning it just for global namespace.

```cpp
#include <iostream>

namespace some
{
    int x{ 0 };
    // some::x cannot be modified in the `some` namespace after definition

}

// some::x cannot be modified directly from the global namespace
// since no statements except declaration and definition are allowed in global namespace
// some::x = 1;


int main()
{
    some::x = 1;    // some::x can be modified from main or any other function
    std::cout << some::x;
    return 0;
}
```

0 Reply

Alex Author

Reply to tatadott[16] November 17, 2023 5:28 pm

Thanks for the suggestion. Tweaked some wording and added an insight box to reinforce this.

0 Reply

park

October 19, 2023 1:06 am

Dear author.
I would like to leave here a humble suggestion, sharing my reading experience.
In the subsection **The global namespace**, it says:
"Only declarations and definition statements can appear in the global namespace. (...) This also means that other types of statements (such as expression statements) cannot be placed in the global namespace (initializers for global variables being an exception):"
I got stuck here for a while, because it implies that "initialization" belonged to "other types of statements".
To me it was not easy too comprehend, because I couldn't think of an example of initialization that is not a definition or a declaration.
Now that I Googled some other materials such as this stack overflow thread (https://stackoverflow.com/questions/23345554/what-distinguishes-the-declaration-the-definition-and-the-initialization-of-a-v)[17], and found your reply in section 2.7 which goes "Initialization is giving the variable an initial value upon creation." (Reply to Serif, October 2, 2023), it seems to me that every initialization is a definition, just like every definition is a declaration.
Is that correct?
If so, I think it would be rather easier to follow if you clarified that in section 2.7 or in this section.

Last edited 6 months ago by park

0 Reply

Alex Author

Reply to park[18] October 20, 2023 2:03 pm

I removed the "(initializers for global variables being an exception)" because it's misleading as written -- initializers are expressions, not statements!

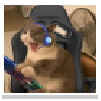
Initialization isn't a definition or a declaration -- initialization is an optional part of a definition.

2 Reply

Yopi
September 30, 2023 4:29 am

Thanks for the lesson

0 Reply



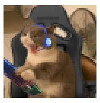
bigbawlsbobby69
May 18, 2023 9:52 pm

I did a little bit of testing, and using the `using namespace` directive within a function works, and appears to not go outside the local scope of the function. Would this be a use case where you could use this directive?

Thanks for any help! :)

Last edited 11 months ago by bigbawlsbobby69

1 Reply



bigbawlsbobby69
Reply to bigbawlsbobby69 19 May 18, 2023 10:30 pm

Just realized that if you order the `cout()` function before you use the `using namespace` directive within another function, it doesn't work. I initially ordered it so that `cout()` was after, so it compiled just fine.

The version that runs fine:

```
#include <iostream>

void print()
{
    using namespace std;
    cout << "Hey all!\n";
}

void cout()
{
    std::cout << "Other hey all!\n";
}

int main()
{
    print();
    cout();

    return 0;
}
```

The version that fails to run:

```cpp
#include <iostream>

void cout()
{
    std::cout << "Other hey all!\n";
}

void print()
{
    using namespace std;
    cout << "Hey all!\n";
}

int main()
{
    print();
    cout();

    return 0;
}
```

It also fails if you use a forward declaration. Guess this would be another point against using the `using namespace` directive unless you're very specific about ordering your functions. Honestly seems like too much of a hassle with projects that use more than one or two user-defined functions, so I likely won't be using it at all in the future.

Last edited 11 months ago by bigbawlsbobby69

6 Reply

Emeka Daniel

Reply to bigbawlsbobby69 20 August 3, 2023 5:26 am

Permit me to Chip in pls:
First of all i want to say something: In C++ we have something called function overloading that lets us define multiple functons with the same name as far as a condition is met. The reason this user defined cout() causes the conflict of ambiguity is because function overloading occurs between functions only not functons and objects. The cout in the std namespace is an object of type ostream, while the cout in our main.cpp, is a function. The compiler is producing an error because it has a problem resolving which identifier we mean to reference, whether it is the cout object identifier that is now in the global namespace due to the using directive or whether it is the cout function identifier that we have defined ourselves. Assuming the cout that lives in the std namespace was a function, it would have been an overload[Assuming you meet said conditions].[The Author Covers Function Overload Later]

Now, what you said above got me intrigued:

Your first example:

```
#include <iostream>

/* This Code works because of the ordering of the functions.
Since all functions have local scope
the using directive in the print() is limited only to the
print(), so calling your user defined cout() function in
function main() does not cause ambiguity*/
void print()
{
    using namespace std;
    cout << "Hey all!\n";
}

void cout()
{
    std::cout << "Other hey all!\n";
}

int main()
{
    print();
    cout();

    return 0;
}
```

Your second example:

```
#include <iostream>

/* Now that you have changed the ordering of the functions,
the compiler has a reason to complain because in function
print(), the using directive gave you access to cout without
the std, and you already have a user defined cout function
identifier above you...So due to the fact that functions can
call other functions, there would be a naming conflict if you
decide to use cout, that lives in the std namespace,
unqualified. If function calling other functions weren't
possible in c++, then this would have been a non-issue.*/
void cout()
{
    std::cout << "Other hey all!\n";
}

void print()
{
    using namespace std;
    cout << "Hey all!\n";
}

int main()
{
    print();
    cout();

    return 0;
}
```

Thanks for the voicing out, your observation made me clarify something.

10 Reply

Anything

Reply to Emeka Daniel [21] October 13, 2023 2:49 am

Hey , thanks for your explicit explanation .

But even with the first version of that code in vs code,where

```
void print()
```

is above void cout() that the "cout under print function is ambiguous.

```
#include <iostream>

void print()
{
    using namespace std;
    cout << "Hey all!\n";    //it says the cout here is
ambiguous
}

void cout()
{
    std::cout << "Other hey all!\n";
}

int main()
{
    print();
    cout();

    return 0;
}
```

what could be causing this? cause we havent define cout until after the print function.

Last edited 6 months ago by Anything

0 Reply

Emeka Daniel

Reply to Anything [22] October 13, 2023 1:45 pm

Yh it does, it's the C/C++ extension that provides intellisense for your code that is complaining.
The code will still compile fine. Try it!

0 Reply

For
May 14, 2023 12:18 pm

So I understand you shouldn't include `using namespace std;` because of potential collision errors but can you provide another example than `cout` conflicting with another `user defined cout`? I'm curious on what could be a more common name collision if you don't include a `user defined cout`.

1 Reply

Alex Author
Reply to For [23] May 14, 2023 9:37 pm

I'd guess `std::max`, `std::sort`, and `std::swap` are all easy targets for name collisions.

6 Reply

For
Reply to Alex [24] May 15, 2023 10:41 am

Ah thank you all the examples we see are of `std::cin`, `std::endl`, and `std::cout` so I wasn't sure what else could conflict I tried looking up the std library but all I got were different header types.

Last edited 11 months ago by For

0 Reply

noctis
March 27, 2023 12:38 am

```
On the other hand, if you had to deliver mail to two addresses, one at 209
Front Street in Mill City, and another address at 417 Front Street in
Jonesville, there would be no confusion about where to go.
```

Shouldn't `417` be `209`.

2 Reply

Alex Author
Reply to noctis [25] March 29, 2023 9:41 pm

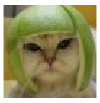
Could be, but isn't necessary. The city is enough to disambiguate which Front Street is intended.

1 Reply

Krishnakumar
Reply to Alex[26] August 25, 2023 7:55 am

Looks like these precise addresses were removed from the lesson?

0 Reply


Alex Author
Reply to Krishnakumar[27] August 28, 2023 1:48 pm

From the lesson:
> On the other hand, if you had to deliver mail to two addresses, one at 209 Front Street in Mill City, and another address at 417 Front Street in Jonesville, there would be no confusion about where to go.

Are you seeing something different than me?

2 Reply

Krishnakumar
Reply to Alex[28] October 10, 2023 11:42 am

There was also 245 Front Street in Mill City at the start of the lesson, which confuses a bit.

0 Reply

Alex Author
Reply to Krishnakumar[29] October 11, 2023 3:37 pm

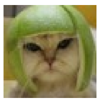
Updated 209 Front Street to 245 Front Street to reduce the number of different addresses floating around.

1 Reply

Suku_go
March 25, 2023 3:07 am

Is compiler error is all the error that being show directly in the editor (the red mark in Visual Studio when we do something wrong) ? Just to be clear.

0 Reply


Alex Author
Reply to Suku_go[30] March 25, 2023 6:47 pm

It's the error that the compiler generates when you actually compile your program.

Some IDEs will add red marks if they detect that something is syntactically invalid. This isn't a compile error, but rather an IDE feature to help you identify issues before you compile your program.

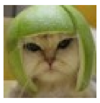
1 Reply

newtocoding
January 18, 2023 1:59 am

```
#include <iostream> // imports the declaration of std::cout

using namespace std; // makes std::cout accessible as "cout"

int cout() // defines our own "cout" function in the global namespace
{
    return 5;
}

int main()
{
    cout << "Hello, world!"; // Compile error!  Which cout do we want here?
The one in the std namespace or the one we defined above?

    return 0;
}
```

we are using here `<<` outputoperator* so why our compiler confuses it with the function "cout" does that mean we can use "<<" operator with other functions also? // i know it is a very dumb question, i hope you don't mind :)

2 Reply



Alex Author
Reply to newtocoding [31] January 18, 2023 3:42 pm

`cout` and "Hello, world!" are the operands to `operator<<`. The compiler can't determine which `cout` it should use.

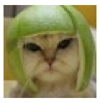
Yes, we can use `operator<<` with other std::ostream objects (e.g. std::cerr). We can also write our own overloaded `operator<<` to work with whatever operand types we desire.

2 Reply

Chris
January 10, 2023 12:05 pm

Question for you Alex: Do you have a project(s) recommendation (or maybe multiple) where I can apply the knowledge I'm learning? I have end-of-course ideas (Graphics programming and/or Audio Plugin development) but I struggle to develop ideas to apply what I'm learning here effectively. And most of all, thank you so much for this site. You have no idea how much it means to me!

2 Reply


Alex Author
Reply to Chris 32 January 13, 2023 12:05 pm

In these early lessons, we simply haven't covered enough fundamentals to build anything particularly interesting. Once we cover random numbers and loops, the possibilities for doing neat things really expand.

The quizzes provided at the end of various lessons are designed to help apply knowledge, but they're fairly primitive.

As a starter, think about the games you know (card games or guessing games) and whether you could take an initial / partial shot at implementing those somehow. If you skip ahead to the lesson on random numbers, you might be able to implement a crude version of a simple game like rock paper scissors.

10 Reply

stu
Reply to Chris 32 January 12, 2023 9:18 pm

I second this question. I feel as though I'm getting it but failing to see the application of what I'm learning. I figure probably just not far enough along in the process yet.

2 Reply

denmeeez
December 27, 2022 4:08 am

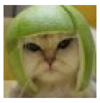
'id returned 1 exit status'

what is this problem?

everything worked and completed okay before this moment..

Last edited 1 year ago by denmeeez

0 Reply


Alex Author
Reply to denmeeez 33 January 1, 2023 6:48 pm

ld is the linker, so the linker is failing to produce an updated executable. Most likely this is happening because the existing executable is running in the background, and it can't be replaced while it is running.

2 Reply

Matthew
December 9, 2022 2:07 pm

So I'm wondering is it possible to create a project in C++ and then make a winform for it in C#? also do we talk about imgui at all in this? Also do we talk about how to make drivers at all?

0 Reply

Alex Author
Reply to Matthew[34] December 10, 2022 1:51 pm

1. I know there is some way for C++ and C# code to interoperate, but I don't know how that works.
2. This tutorial series generally doesn't talk about 3rd party libraries, as those come and go. We're more focused on core language concepts.
3. No, as that's an OS/hardware-specific topic.

3 Reply

Anonymous
November 9, 2022 7:02 am

Unless there's an actual occurrence where you'd want to use specified keywords of a namespace as variable / function names, I don't see why we shouldn't use namespaces. They make the code look much better and less daunting for newcomers.

1 Reply

slashtab
Reply to Anonymous[35] November 12, 2022 9:23 am

I think it's better to use explicit namespace because beginner or not you don't need wonder about if the identifier is user-defined or a namespace one and it is also about developing habit to write explicit namespace, which will work in long run.

4 Reply

hantao
October 30, 2022 6:34 pm

but this can be used in a same file, say they are two different functions, right?

```
int add(int x, int y, int z)
{
    return x + y + z;
}

int add(int x, int y)
{
    return x + y;
}
```

0 Reply

glokmw2
Reply to hantao 36 November 3, 2022 2:19 am

I might be wrong but this is good example of function overloading?

3 Reply

slashtab
Reply to glokmw2 37 November 12, 2022 9:26 am

Yes!!

1 Reply

-ACK
Reply to hantao 36 October 31, 2022 9:02 am

Yes, because they have distinct parameters

4 Reply

naming collision
September 5, 2022 7:45 pm

>>2. Two (or more) definitions for a function (or global variable) are introduced into the same file (often via an #include). This will result in a compiler error.

I was wondering why on the previous lesson when we included "#include "iostream"" twice in main.cpp and input.cpp it didn't throw any naming collision?

2 Reply

#include hater
Reply to naming collision 38 October 21, 2022 4:57 am

I'd recommend not using #include at all and write your own code that does the function for you. This could take a while but would save you from having to use those nasty include statements.

0 Reply

Tom Duhamel

Reply to naming collision [38] September 15, 2022 9:53 pm

<iostream> does not contain any definition at all, only declarations. You will find that decorations can be duplicated as often as necessary. There will not be any conflict, as long as all of the different declarations are of the same signature.

1 Reply

Alex Author

Reply to naming collision [38] September 13, 2022 5:34 pm

Header guards (which every header file should have) will prevent the content of a header file from being #included more than once into the same file.

4 Reply

slashtab

Reply to Alex [39] November 12, 2022 9:42 am

In the Tom Duhamel comment here (https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-573210)[40] he is suggesting that <iostream> has only declaration so we don't need to worry but then I read your comment and got confused between two. My question is do we need Header guards in declaration files too? header guards in iostream is preventing it from getting included twice or not?

0 Reply

Alex Author

Reply to slashtab [41] November 13, 2022 6:08 pm

I think Tom is incorrect, as iostream contains template class definitions (or includes other headers that do).

All standard library headers will have header guards to prevent them from being compiled in a translation unit more than once.

2 Reply

naming collision

Reply to Alex [39] September 13, 2022 5:50 pm

It means both #include "iostream" has a header guard! That makes sense. Thanks!

Last edited 1 year ago by naming collision

0 Reply

Stan

Reply to naming collision 38 September 6, 2022 8:02 am

The answer is in your comments I guess?
if #include is in the same file.. compile error. But we used it in two different files (main.cpp and input.cpp)
Without #include <iostream> you can not access the input/output functions we are using :)

0 Reply

bpw

Reply to Stan 42 September 7, 2022 2:24 am

Because iostream is an include that contains the function declarations only. Each .cpp file needs those declarations in order to compile. Remember that each .cpp file is compiled as a standalone unit.
The linker finally links everything against the C++ standard library which contains the definitions of the functions declared in the iostream header. Thus, there are no multiple function definitions.

0 Reply

naming collision

Reply to Stan 42 September 6, 2022 10:15 am

I am afraid you're wrong. If we #include the same file twice in a file, no naming collision will occur and that makes me confused why?!

0 Reply

Stan

Reply to naming collision 43 September 6, 2022 10:21 am

I guess I am way too 'begginer'to comprehend your question properly. I just started this course, but still feel like CPP being 'trust the programmer'language is what makes this possible. Since it reads each file independently it just trusts you that you know what you are doing. Sorry if I still can not properly comprehend your question but I hope I will get something out of it too if that is the case :)

Last edited 1 year ago by Stan

0 Reply

Austin
August 14, 2022 6:11 pm

This is entirely in my opinion, I think adding alternate definitions next to keywords would be very beneficial for someone like me, however, if you don't feel like doing this, anyone can just write the definitions down in their notes and look at it when they need to.

0 Reply

Alex Author
Reply to Austin 44 August 15, 2022 11:45 am

What do you mean by "alternate definitions"?

1 Reply

Dave
Reply to Alex 45 August 16, 2022 6:43 pm

I'm just extrapolating, but OP might mean something like a paragraph of an explicit definition next to (maybe a hover-over) those bolded key terms such as **naming collision** or **the global scope**. I could be wrong though. Awesome guide nonetheless!

0 Reply

Krishnakumar
July 20, 2022 12:25 pm

>We talk more about using statements (and how to use them responsibly)

Potentially confusing tautology.

Perhaps reword the parenthesised text as "and how to employ them responsibly"

0 Reply

Cris
Reply to Krishnakumar 46 October 25, 2022 7:06 am

No, this really isn't confusing at all.

3 Reply

Krishnakumar

Reply to Cris [47] August 25, 2023 8:04 am

Yes. Seems reasonably easy to understand when I read it again today.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/
3. https://www.learncpp.com/cpp-tutorial/why-non-const-global-variables-are-evil/
4. https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/
5. https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/
8. https://www.learncpp.com/naming-collisions-and-an-introduction-to-namespaces/
9. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-594777
12. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-593871
13. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-592681
14. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-589704
15. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-589798
16. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-589916
17. https://stackoverflow.com/questions/23345554/what-distinguishes-the-declaration-the-definition-and-the-initialization-of-a-v
18. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-588939
19. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-580515
20. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-580517

21. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-585025
22. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-588611
23. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-580366
24. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-580396
25. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-578713
26. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-578768
27. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-586109
28. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-586292
29. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-588455
30. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-578670
31. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-576349
32. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-576051
33. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-575587
34. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-575164
35. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-574567
36. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-574353
37. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-574470
38. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572748
39. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-573085
40. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-573210
41. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-574637
42. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572758
43. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572762

44. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572058
45. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572114
46. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-570858
47. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-574213